Identifying and Describing Information Seeking Tasks

Satterfield, Chris ; Fritz, Thomas ; Murphy, Gail C

Abstract: A software developer works on many tasks per day, frequently switching between these tasks back and forth. This constant churn of tasks makes it difficult for a developer to know the specifics of when they worked on what task, complicating task resumption, planning, retrospection, and reporting activities. In a first step towards an automated aid to this issue, we introduce a new approach to help identify the topic of work during an information seeking task — one of the most common types of tasks that software developers face — that is based on capturing the contents of the developer's active window at regular intervals and creating a vector representation of key information the developer viewed. To evaluate our approach, we created a data set with multiple developers working on the same set of six information seeking tasks that we also make available for other researchers to investigate similar approaches. Our analysis shows that our approach enables: 1) segments of a developer's work to be automatically associated with a task from a known set of tasks with average accuracy of 70.6%, and 2) a word cloud describing a segment of work that a developer can use to recognize a task with average accuracy of 67.9%.

DOI: https://doi.org/10.1145/3324884.3416537

Identifying and Describing Information Seeking Tasks

Chris Satterfield
University of British Columbia
Vancouver, Canada

Thomas Fritz
University of Zurich
Zurich, Switzerland

Gail C. Murphy
University of British Columbia
Vancouver, Canada

ABSTRACT

A software developer works on many tasks per day, frequently switching between these tasks back and forth. This constant churn of tasks makes it difficult for a developer to know the specifics of when they worked on what task, complicating task resumption, planning, retrospection, and reporting activities. In a first step towards an automated aid to this issue, we introduce a new approach to help identify the topic of work during an information seeking task — one of the most common types of tasks that software developers face — that is based on capturing the contents of the developer's active window at regular intervals and creating a vector representation of key information the developer viewed. To evaluate our approach, we created a data set with multiple developers working on the same set of six information seeking tasks that we also make available for other researchers to investigate similar approaches. Our analysis shows that our approach enables: 1) segments of a developer's work to be automatically associated with a task from a known set of tasks with average accuracy of 70.6%, and 2) a word cloud describing a segment of work that a developer can use to recognize a task with average accuracy of 67.9%.

CCS CONCEPTS

• **Human-centered computing → Empirical studies in HCI**.

KEYWORDS

software development productivity, information seeking tasks

ACM Reference Format:
Chris Satterfield, Thomas Fritz, and Gail C. Murphy. 2020. Identifying and Describing Information Seeking Tasks. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416537

1 INTRODUCTION

Developers work on many tasks in a day: some of of these tasks are code-related and others involve information seeking [20]. As developers work they switch between tasks constantly [7, 17]. This constant switching, and the variety and high number of tasks, make it difficult for developers to know which task they worked on when. As a result, developers spend significant time and effort recalling what information is needed when a task is resumed [12, 26, 27]. Additionally, developers are unable to accurately record how much time is spent on tasks, impacting personal planning and retrospection activities (e.g., [16]), as well as impacting effort estimation for the entire team.

To ease this problem, some developers, manually track and note which information they access while performing a task as a form of externalization of the working state of a task [27]. This manual approach is time consuming and requires substantial effort from the developer. Some tools have been introduced to alleviate parts of this burden from the developer. For instance, the Mylyn tool enables a developer to indicate when work on a particular task is started and stopped, and the tool then tracks relevant information for the task [12]. All of these approaches require the developer to explicitly indicate *when* they start working on a task, and *which* task they are working on, which is cumbersome at best.

Recently, researchers have made increasing progress on automatically identifying *when* developers switch tasks [13, 18, 21, 25, 31, 32]. These advances mean it is becoming possible to automatically split a developer's past work into segments associated with different tasks. An open problem is to determine *which* task a developer is working on and associating each segment with the task.

In this paper, we explore this open problem, focusing on whether the topic of work—a task—can be identified automatically based on the information that a developer accesses as part of a task. Our initial focus is on information seeking tasks. As it is common in software development to record tasks to be performed in either a shared or private issue repository, we first assume that descriptions of what work is or has been performed are available and examine the following research question:

RQ1: Can we automatically associate existing task descriptions with information developers access as they work on these tasks?

Approaches that address this question can help a developer to locate when they had performed work on a particular task. In a second step, we examine whether it is also possible to generate task descriptions from scratch based solely on the performed work:

RQ2: Can we automatically create a word cloud representation of work performed that enables developers to identify the task on which work was occurring?

To explore these questions, we developed an approach that generates representations of a developer's work for a given time period. The approach, as depicted in Figure 2, takes in work being performed by a developer. Specifically, our approach continuously records screenshots of the developer's active window and utilizes optical character recognition (OCR) to extract the information from it. For a given time period, the approach then applies natural language processing and information retrieval techniques to generate a vector representation of the segment. This representation can then be matched to existing task descriptions to determine the task the developer worked on (for RQ1) or can be used to generate a

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09.
https://doi.org/10.1145/3324884.3416537

word cloud that highlights the most relevant words to describe the task (for RQ2). An advantage of our approach using screen shots and OCR is that it is agnostic to the applications a developer uses to perform their work.

We performed two evaluations to assess our approach: one for each research question. These evaluations are based on a data set that we created consisting of work streams from 17 participants experienced in software development performing six information seeking development-oriented tasks in an interleaved fashion in a controlled lab setting (Section 3). We designed the tasks to be representative of information seeking tasks commonly performed by software developers [6]. Based on manually identified task switches, we then apply our approach on each segment of work (between task switches) to generate vector representations and word clouds for each task segment (Section 2). As we show in Figure 1, we evaluate RQ1 by examining whether our approach can correctly associate task descriptions written by various developers with the task segments using the generated vector representations. We gathered task descriptions for each task from 20 people experienced in software development and found that our approach is able to correctly associate the descriptions with the correct segment of work in 70.6% of cases (Section 4). We perform a preliminary evaluation of RQ2 by examining whether software developers are able to match the generated word clouds to the corresponding tasks. We surveyed 28 experienced software developers and found that they were able to match the word clouds to the six original task descriptions correctly in 67.9% of cases (Section 5).

This paper makes four contributions:

- A data set from a controlled lab setting involving 17 participants working on six information seeking development tasks; other researchers can build on this data set to investigate other approaches.
- An application-agnostic approach to generate representations of a developer's work for a given time period to help determine and describe the task that is being performed and an evaluation of a variety of techniques for generating these representations.
- An evaluation of the approach's accuracy for determining the task a developer was working on for a given time period, based on the collected data set and task descriptions from 20 participants.
- An evaluation of the approach's ability to generate word clouds for task segments that can be used to identify the tasks developers were working on.

While our evaluation only focuses on information seeking development-oriented tasks and uses recorded task switch information, the results show promise for our approach's ability to automatically identify and describe the tasks a developer is working on and for further automating task support.

2 GENERATING TASK REPRESENTATIONS

Our goal is to create representations of a developer's work that allow the developer to determine the tasks worked on. Specifically, we consider the creation of two representations of work performed: a vector space representation (vectors) that can be used to automatically match it to existing task descriptions and thus determine the task worked on; and a word cloud representation that describes the task and allows the developer to identify the task worked on without pre-existing task descriptions. Previous work has shown that word clouds are useful aids for helping users determine the relevance of a document to a topic [8].

We describe our approach that continuously monitors a developer's work by recording screenshots of the active windows, processes and extracts relevant information, and is able to generate vectors and word clouds for specified time periods of work. By recording and processing screenshots, our approach is agnostic to the applications developers use for their work. Figure 2 depicts the main steps involved in our approach.

For this research, we focus on generating representations for *task segments*—time periods of work in which a developer works on one task before switching to another one—and assume that these switches can automatically be determined using emerging techniques (e.g., [13, 18, 21, 25, 31, 32]).

2.1 Screenshot Pre-processing

Our approach first prepares the recorded screenshots of developers' active windows for the optical character recognition (OCR) with Tesseract [34]. Specifically, we convert the colored screenshots to grayscale and scale the resolution down to 300 DPI. These steps are considered best practice as Tesseract was originally intended for reading black on white paper documents. In addition, we crop a percentage of the top of the screenshot as most application windows have menu or bookmark bars at the top that generally do not contain information specific to the task at hand. Through experimentation, we found that removing the top 15% of the screen across all application window screenshots provides a good balance between removing noise without much loss of meaningful content.[1] We automate all screenshot pre-processing steps with the ImageMagick tool [10].

2.2 Extracting Bags of Words

After pre-processing, our approach applies the Tesseract OCR engine to extract the textual content of each screenshot. As Tesseract tries to preserve the format of the text, it produces a structured string for each screenshot. We store these strings in a document, one for each screenshot. These structured strings contain substantial noise even after the pre-processing. For instance, an 'I' is often misinterpreted as the number '1' or the letter 'l'. As well, many nonsensical artifacts can be produced due to noise from items like images and menu bars on the screenshot that remain after pre-processing.

To break these documents into usable pieces of information (words/tokens) and further reduce noise, our approach supports the application of one of two techniques, either (a) tokenization, or (b) keyword extraction. For tokenization, we use the Natural Language Toolkit (NLTK) [24] Version 3.2.5 and apply standard word tokenization techniques based on white space and punctuation to generate lists of all words in a screenshot. We further remove all stop-words from the lists. For the keyword extraction, we use an open source implementation [29] (version 1.0.4) of the RAKE algorithm [30]. Based on an input string, RAKE produces a set

[1] This percentage might have to be adjusted for different screen resolutions and sizes.

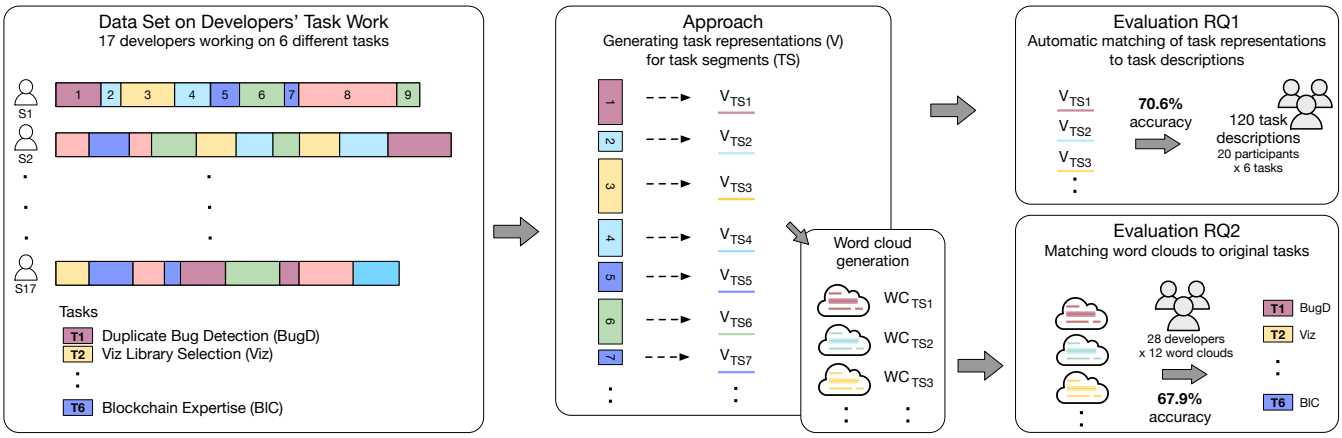

Figure 1: Overview of the process used to evaluate our approach.

Table 1: Techniques used in the vectorization process of a task segment.

Technique	Description
TF	Uses the frequency of a term/word in a task segment as vector entry, and 0 if the word does not occur in the task segment. The vector dimensions are the unique words that occur in any of the task segments for a developer.
TF-IDF	Has the same vector dimensions as TF, but uses TF-IDF for calculating entries. TF-IDF for a word/term t is defined as $f_t * idf_t$, where f_t is the term frequency of t, and idf_t is calculated as $log \frac{N}{df_t}$, where N is the total number of task segments and df_t is the number of task segments in which t occurs.
W2V	Uses a word2vec [19] model pretrained on a corpus extracted from Wikipedia. Each word within a task segment is assigned a 300 dimensional embedding vector. These vectors are then averaged to create one embedding vector for the entire task segment. This technique has been shown to be an effective baseline in many NLP tasks [11].

of keywords with a size equal to 1/3 of the number of original words (not counting duplicates). After breaking up each document into a set of words using tokenization or keyword extraction, our approach stems all words using the Porter stemmer implementation from NLTK.

Finally, the approach creates a bag of words—a record of the frequency of each word—for each task segment by aggregating all words extracted from all screenshots of a task segment.

2.3 Generating Task Representations

A bag of words is itself a primitive representation of a task with the frequency of each word in the bag indicating the importance of a word to the task. However, this representation only reflects importance of a word with respect to the current document (task segment). To also take into account the relevance of the word in context of the overall work of the developer and further help filter noise from the screenshots and the OCR, our approach is designed to enable experimentation with several natural language processing (NLP) and information retrieval techniques to generate more advanced representations.

Table 1 summarizes the three techniques we experimented with in this work to produce (a) a vector space representation V, and subsequently (b) a word cloud representation WC. Each of these techniques takes the words from the bag of words produced in the previous step as input, and produces a vector representation of the task segment. In the case of *TF* and *TF-IDF*, the dimension of the vectors is the number of unique words in the set of all words from all task segments of a developer. For *W2V*, we chose the dimension of the vectors to be 300 based on our training of the word2vec model. All of these vector representations can then be compared using cosine similarity against vectors which could be generated based on other task segments or, for example, task descriptions.

Based on these vector representations, our approach can be used to generate word clouds. However, since the meaning of the dimensions in the W2V vectors are difficult to interpret, we did not use the *W2V* technique for creating word clouds. To generate word clouds for the *TF* and *TF-IDF* technique, our approach selects the 100 largest entries in a vector, corresponding to the highest ranked words in a task segment, and use the score of the words to determine the proportional size of the words in the cloud. Figure 3 shows two examples of such word clouds.

3 DATA SET CREATION

To support the investigation of the two research questions, we created a data set from 17 developers working in a controlled laboratory setting on a set of six information seeking tasks over a 2 hour time period. We chose a laboratory setting to be able to gather

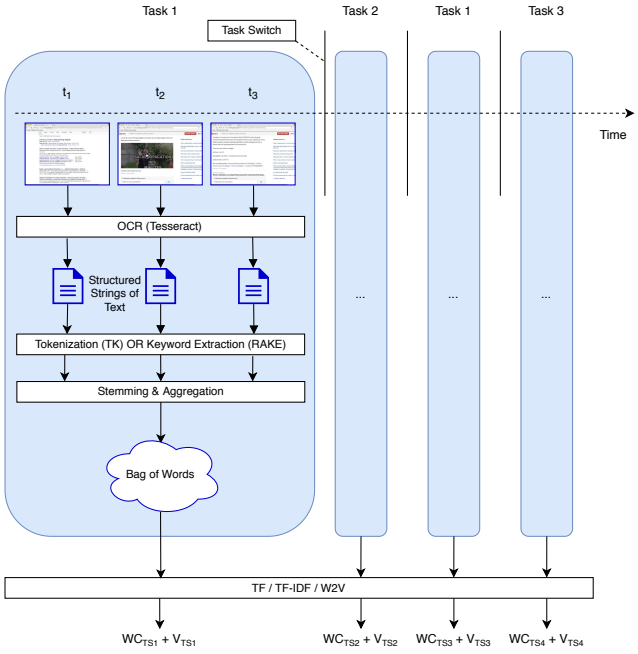

Figure 2: Main steps of the approach to generate task representations (the light blue boxes represent task segments).

(a) WC for a Deep Learning Presentation (DeepL) task segment of D1

(b) WC for a Duplicate Bug (BugD) task segment of D4

Figure 3: Word Clouds (WCs) generated for task segments from different tasks and developers.

data from multiple developers working on the same tasks. The full data set will be made available in the supplementary material[2] [1].

[2]The data set is temporarily withheld to protect double blind during the review process.

3.1 Developers

We recruited 17 participants—who we refer to as *developers* in the following—through advertising at our university and personal contacts. All developers had several years of experience in software development, with an average of 6.4 (±2.4) years per developer. 10 of the developers were female, and 7 were male. At the time of the data set creation, 10 were graduate students, 4 were upper year undergraduates, and 3 were interns at a mid-sized software company. All developers were residents of Canada.

3.2 Tasks

Developers work on many different kinds of tasks each day, some of which focus on code (23.4% [17]) and some of which focus on information seeking (31.9% [6]). In collecting this dataset, we chose to focus on information seeking tasks. We made this choice given the significant, and higher, fraction of their day developers spend on these kinds of tasks. This choice also enabled developers to attempt more tasks in the limited two hours available per developer; coding tasks would have required more time per developer to enable developers to gain sufficient familiarity with a codebase. We discuss the implications of our choice in focusing on information seeking tasks in Section 6.1.

We created six tasks that are representative of common information seeking tasks based on the authors' knowledge of industrial development. The tasks we selected were designed to be realistic, yet simple enough for it to be possible for developers to make significant progress in the limited time available. The tasks were also chosen to enable a developer to make progress without prior knowledge. Developers were not constrained in how they approached a task.

Table 2 provides a short description of each of the six tasks, including a short name that we use in this paper to refer to a specific task; the short task name and description was not presented to developers. An example of one of the actual task descriptions used in this study is presented in Table 3. We intentionally designed the `App Market Research Task` and `Recommend Tool Task` as tasks which were likely to have very similar information accessed as part of working on the task to allow us to assess the discriminative power of our approach. Full descriptions of the tasks the developers worked on can be found in the supplementary material [1].

3.3 Session

Before the start of a session, we gave each developer a brief overview of the procedure they would be asked to follow. Developers were told that they would be asked to work on a number of information seeking tasks, and that they could accomplish these tasks in whatever manner they chose. However, the quantity of tasks and the content of each task was withheld until the session commenced. Developers were also told that their screen would be recorded by our monitoring tool, and that they would be observed by the observer as they worked.

At the start of a session, developers were presented with a list of 6 tasks to perform within a 2 hour time period. The tasks were presented in the form of unread emails sitting in an inbox accessed by a webmail client. The order in which the tasks appeared in the inbox for a developer was randomized. We asked a developer to

Table 2: Overview of Controlled Lab Tasks.

Abbrev.	Short Task Name	Short Task Description (by us)
BugD	Duplicate Bug	Examine a collection of bug reports from a Bugzilla repository to determine if any were duplicates. (Each developer was asked to examine four bug reports, two being duplicate reports and two not. For each participant, bug reports were randomly selected from the set of all resolved bug reports from the Mozilla projects (e.g., Firefox, Thunderbird, etc.) [23] over the course of a month.
Viz	Viz Library Selection	Research visualization libraries and identify one which is suitable for outlining the benefits of your companies tool, for creating a presentation to clients.
PrMR	App Market Research	Perform market research on three productivity apps. Identify common functionalities, similarities and differences, and report on your findings.
PrRec	Recommend Tool	Examine app store reviews for three productivity apps (the same ones as above) in order to recommend one to your coworker.
DeepL	Deep Leaning Presentation	Prepare in advance answers to likely questions for a hypothetical presentation you are giving about potential deep learning applications.
BlC	Blockchain Expert	Answer your coworkers follow-up questions about a hypothetical presentation you gave about the different ways your company could make use of blockchain.

Table 3: Full Task Description for the App Market Research Task (PrMR) as Presented to Developers.

The software company you work for is considering expanding into the productivity tool sphere. Your manager has asked you to do some market research on 3 of the most popular already existing apps in this domain: Microsoft To-do, Wunderlist, and Todoist. Provide a short written summary of the similarities and differences between these 3 apps.

work on the tasks on a laptop with a *13.3 inch, 1440x900* sized screen running macOS which was instrumented with our recording tool (reference omitted for double blind). As a developer worked on the tasks, the tool recorded screenshots of the developer's active window at 1 second intervals. Application names and window titles were also recorded whenever they changed.

To simulate interruptions, we also installed a tool on the laptop that produced a popup in random intervals lasting from 6.5 to 16.5 minutes. The average time between popups was selected as 11.5 minutes, in accordance with González and Mark's findings on the average amount of time knowledge workers spend in a working sphere segment before switching [7]. To simulate the disruptive effects of a real external interruption, the popup prompted developers to solve an arithmetic question before switching to a new task. These popups were excluded from our tools recordings to avoid biasing our results.

As a developer worked on the tasks, a researcher manually annotated the times at which the developer switched tasks, also keeping track of the task the developer was working on. After the session was complete, the times at which switches happened were verified and adjusted by reviewing a screen capture that ran in the background of the provided laptop, to ensure task switches were recorded accurately.

3.4 Data Collected

In total, we were able to collect screenshot data for all 17 developers and on all six tasks for each developer. All but one developer completed the 6 tasks within the allowed time period. On average, developers took 91.2 ±17.5 minutes to complete the six tasks and we collected an average of 5131 screenshots per developer. Due to a technical issue, we were able to gather window titles for only 12 of the 17 developers. The full data set will be made available in the supplementary material[3] [1].

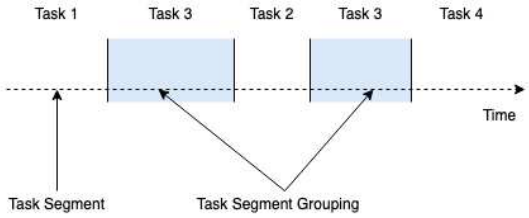

Figure 4: An example of a developer working on several tasks over time, revisiting task 3 in two task segments.

3.5 Data Annotation

Using the task annotations collected by the researcher during each session, we annotated the collected data with the task switches and the task the developer was working on. Each session resulted in the developers working in an interleaved fashion on the six tasks. Figure 4 depicts a portion of a developer's work, showing an example of the interleaving. We define a *task segment* as the period of time between two task switches, during which a developer was working on a specific task. We define a *task segment grouping* as the collection of all task segments that collectively represent work on a specific task. We use *task segment groupings* as a baseline for evaluating our approach, as it mimics the simplest case in which each task is completed in one contiguous segment and we have

[3] The data set is temporarily withheld to protect double blind during the review process.

the entirety of the information accessed for the work on a task available.

4 RQ1: IDENTIFYING TASKS

Our first research question asks whether we can automatically associate descriptions of a developer's tasks with the information the developer accesses as she works. Performing this association automatically is challenging because there are many ways in which a developer can complete a task and there are many ways in which a developer can describe the task on which they are working.

The data we collected in the lab setting (Section 3) includes a number of ways in which the tasks assigned could be completed. While participants in the lab setting had some overlap in the resources they accessed as part of a task, no two participants completed a task in exactly the same manner.

Similarly, developers are likely to tailor their task descriptions towards the ways they might approach a task. To study the first research question, we therefore also needed a range of descriptions of the tasks on which the developers had worked. To gather these descriptions, we employed Amazon Mechanical Turk (MT). Given a range of descriptions collected in this way, we are able to assess how the range of techniques we developed for generating task representations (Section 2) can address the first research question.

4.1 Gathering Task Descriptions

To capture a range of task descriptions, we distributed a survey via Mechanical Turk. As a requirement for responding to our survey, we asked that respondents be currently or previously employed in the software industry. In total we received responses from 29 respondents. These respondents represented a range of fluency with English and a range of experience in software development. On average, respondents had 6.2 (±5.4) years of software development experience, and 3.8 (±3.5) years of professional development experience. Of these respondents, 24 reported that they were native English speakers, while 3 reported being fully fluent and 2 reported being proficient.

Respondents of this survey were presented with the same set of six full task descriptions that we also used for the data set creation in the controlled lab setting. An example can be seen in Table 3. We asked respondents to "Please summarize the task described below in your own words, as you might write it for your own reference in a to-do list or similar. Please limit your response to at most 15 words.". Thereby, we randomized the order in which the full task descriptions were presented.

To filter out irrelevant or low quality responses, we asked two external experts, who were both researchers in the software engineering domain and experienced software developers, to rate the quality of every task description generated by each respondent. Each rater was instructed to use a scale from 1-3 to indicate the relevancy and quality of the responses, with a score of 1 indicating an irrelevant response, 2 indicating relevant but low quality responses, and 3 representing relevant and high quality responses. We found that the distinction between responses rated 2 or 3 varied greatly between our two experts, but that there was a strong consensus with regard to the responses which were rated 1 / irrelevant (Cohen's Kappa: 0.74, indicating strong agreement [15]). These irrelevant responses tended to come in multiples from the same participants. We removed all participants with irrelevant responses and considered only those responses which both authors rated with a score of 2 or higher, leaving us with 20 participants and a total of 120 task descriptions. A sample of 3 responses for one task with the ratings by one expert rater is depicted in Table 4.

4.2 Evaluation

From the controlled lab setting, we have 189 task segments and 102 task segment groupings. From the MT survey, we have 20 descriptions for each task, resulting in a total of 120 task descriptions. We wish to determine if the approach we developed for generating task representations, and which choice of techniques within the approach, can be used to determine which task segment (or task segment group) maps to which task description with sufficient precision and recall, even when these task descriptions might vary. Recall that we know the ground truth of which task, and thus which task description, each task segment represents based on notes taken by a researcher during the controlled lab setting.

Our evaluation consists of considering each task segment from a lab developer's work and mapping it to one of the six task descriptions produced by a MT respondent. We use this evaluation method that assumes a complete set of descriptions as we wish to assess how well our approach might work in a situation where a developer may be trying to determine, from a given set of tasks, when they performed work on each task. For the mapping, we generate a vector space representation of the task segment V_{TS} as well as one for each of the six task descriptions V_1 to V_6 produced by a respondent and then calculate the cosine similarities between V_{TS} and each of V_1 to V_6. We choose the task description most similar to our generated task representation and evaluate it by comparing it to the ground truth to determine if it is correct.

For generating task representations from task segments in vector space format, we experimented with and compared six (3 x 2) different combinations of techniques: 3 different techniques for vectorization (term frequency, TF-IDF, and word2vec word embedding), and 2 different techniques for extracting bags of words (tokenization using NLTK, and keyword extraction using RAKE as described in Section 2.2). The vectorization techniques applied are described in Section 2.3.

To generate vectors from the task descriptions of MT workers, we tokenized the task descriptions using NLTK (keyword extraction is not useful in this case given the brevity of the descriptions), and then applied the exact same vectorization technique as used for the task segments, i.e. either TF, TF-IDF, or W2V.

4.3 Results

Figure 5 illustrates the results of the comparison between the six different combinations of vectorization and word extraction techniques. Overall, the combination of TF-IDF with simple word tokenization performed the best, however the differences are small compared to the combination with RAKE or using just TF. Ultimately, word2vec performed the worst for the generation of task representations and mapping to the task descriptions. Since word2vec is also the most computing intensive, it was the least appropriate for this scenario. Based on these results, we selected the combination

Table 4: Examples of descriptions received for the *Viz Library Selection (Viz)* task, together with one of the expert's ratings.

Rating	Task Description (Survey Response)
Irrelevant (1)	I would suggest SIMILE Exhibit or InfoVis Toolkit for Javascript libraries to create a visualization.
Low Quality (2)	Visualize workers work pattern.
High Quality (3)	Create visualizations for product benefits. Select libraries and give existing work examples.

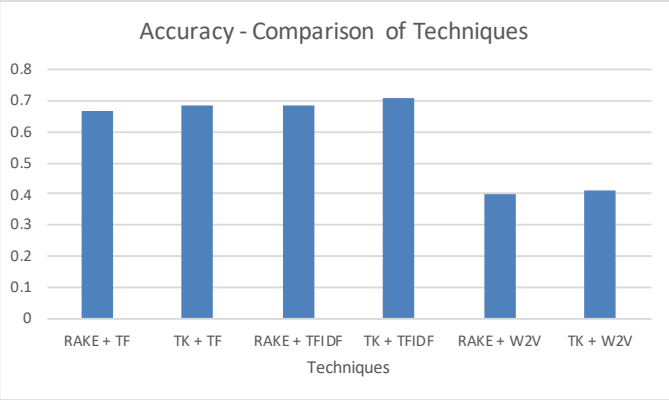

Figure 5: Accuracy comparison for the 6 different combinations of techniques used to generate task representations (TK = tokenziation).

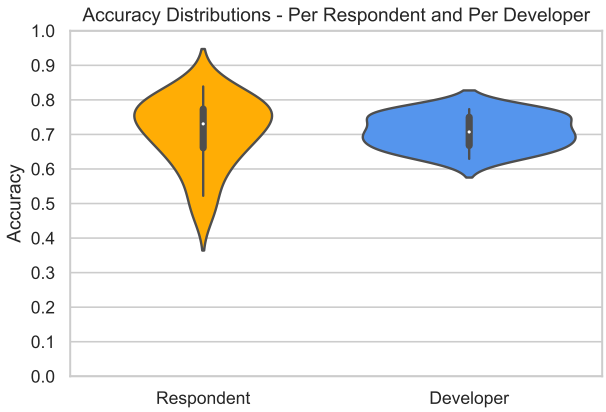

Figure 6: Accuracy distributions by MT respondent and by lab developer for mapping task representations to the correct task descriptions.

of TF-IDF with word tokenization (NLTK) as the approach that we use for the remainder of the paper.

Table 5 presents the results of the evaluation when mapping task representations of task segments (or task segment groupings) to the task descriptions written by the 20 MT workers. We report the precision and recall for each of the 6 tasks, calculated on a per task segment basis (or with the baseline of the per task segment grouping). Overall, using only task segments, our approach achieved high accuracy across all tasks (70.6%) in comparison to a random classifier (16.7%). Accuracy for task segment groupings was moderately better (75.5%). This is a promising result as it indicates that there is often already sufficient information in an individual task segment to predict the task that is being performed; adding more information helps some but does not make a dramatic difference.

Our approach performed well at predicting tasks with a distinct focus, such as DeepL and BlC, with precision values over 80%. This result is unsurprising, as in order to perform these tasks, the developers in the lab setting tended to turn to resources that contained a dense amount of highly specific information related to these topics, such as the Wikipedia pages for blockchain and deep learning. The presence of dense, consistent information eases the production of accurate representations for the tasks. Our approach also performed well at recognizing the BugD task with precision over 92%. We found this result surprising as we expected this task to be one of the more difficult tasks to predict, especially since our summary authors were given no information about the content of this task, beyond that it involved finding duplicate bugs in a Bugzilla repository. As expected, the most difficult to predict tasks were the PrMR and PrRec tasks. These tasks are very similar and as such, resulted in very similar task representations as well as very similar task descriptions and in turn, in a high confusion between the two tasks.

Figure 6 illustrates the accuracy distributions of the results on a per developer and a per MT respondent level. Despite differences in the way each developer performed each task, the results are fairly consistent across developers, ranging from a minimum accuracy of 62.9% to a maximum of 77.3%. Across the MT respondents that authored the task descriptions, the results are mostly consistent, however, there is a significant variation for a few respondents. The respondents for which the accuracy of mapping task representations to their task descriptions were rather low tended to be ones who authored multiple descriptions that were also rated lower by the experts (e.g., item 2 in table 4 was written by author S6). These result demonstrate that the task representations are relatively robust across developers and different ways of performing the tasks, and that writing precise and somewhat detailed descriptions of the tasks being performed clearly impacts the results of our approach.

5 RQ2: DESCRIBING TASKS

To address the question of whether we can automatically generate word cloud representations of information accessed by a developer which would help the developer to identify what task she worked on during a specific period of time, for a task which would help developers identify what task they worked on during a specific period of time, we evaluated the word clouds we generated as described in Section 2.3. We asked 28 participants experienced in software development to match our generated word clouds to the original full task descriptions of the tasks that were performed during the data set creation.

Table 5: Results of mapping task representations to task descriptions written by MT workers.

	Task Segments						Task Segment Groupings					
	BugD	Viz	PrMR	PrRec	DeepL	BlC	BugD	Viz	PrMR	PrRec	DeepL	BlC
Precision	92.6%	72.7%	53.3%	44.7%	83.5%	80.9%	97.4%	78.1%	55.0%	55.7%	85.3%	86.4%
Recall	82.3%	81.5%	47.1%	65.4%	75.3%	70.0%	89.4%	89.4%	45.0%	70.9%	82.1%	76.5%

5.1 Survey

To evaluate the quality of our automatically generated word clouds as a visual representation of a task, we conducted a survey with experienced software developers. Participants were recruited through personal and professional contacts, and as an incentive for responding were entered into a draw for one of two $25 gift cards if they desired. In total, we received survey responses from 28 individuals, with an average of 8.0 (±3.9) years of software development experience. 20 participants were male, while 8 were female. 9 participants reported they were native English speakers, 12 reported that they were fully fluent in English, and the remainder (7) reported that they were proficient in their understanding of English.

We asked our participants to match word clouds to corresponding tasks by presenting them with the list of the six full task descriptions that we also used for the data set creation. An example of one of the descriptions can be seen in Table 3. The word clouds used in the survey were generated following the procedure described in Section 2.3. Using the data that we collected across all 17 developers in the data set creation (Section 3), we randomly selected 4 task segments and 4 task segment groupings for each of the six task and generated word clouds for these, resulting in a total of 48 word clouds. Since asking survey participants to examine a total of 48 word clouds would be too much and impractical, we randomly selected and asked each participant about 12 of the 48, ending up with 2 word clouds (1 for a task segment, 1 for a task segment grouping) for each of the six tasks. Examples of two of these word clouds can be found in Figure 3. We asked participants to read the six full task descriptions and to then identify which task the presented word clouds describe best. Participants also had the option to indicate that the word cloud does not match any task.

5.2 Results

We aggregated the results of the survey responses to obtain accuracy ratings for the word clouds we generated. Overall, the average accuracy of mapping word clouds to the corresponding tasks was 67.9% for the word clouds generated from task segments, and 69.6% for the word clouds generated from groupings. Figure 7 shows the breakdown of the accuracy on a per task level. The success rates of our participants varied widely between tasks. For example, for the blockchain expert task BlC, our participants were able to correctly identify the task for the generated word cloud 100% of the time. Conversely, participants struggled to properly identify the task for the word clouds generated for the duplicate bug task BugD (35.7%). This task was by far the most difficult for participants to identify, and many participants reported that the word clouds generated by this task were not descriptive. Unsurprisingly, participants frequently confused the word clouds generated for the app market research task PrMR and for the recommend tool task PrRec. These

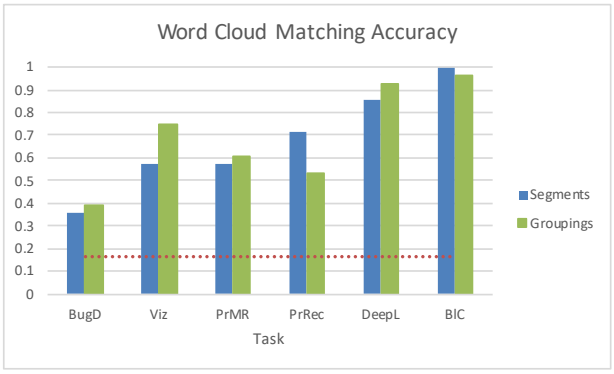

Figure 7: Accuracy for identifying the task based on our generated word clouds. The dotted red line indicates the accuracy for a random classifier.

word clouds tended to have very similar key words, as both full task descriptions mentioned the same three productivity tools.

Comparing the results of the word clouds generated from segments to the ones generated from groupings did not reveal a substantial difference. This is a promising result, as it indicates that enough data can be generated from within the bounds of most task segments to create word clouds that accurately represent the topic of a task as a whole.

6 DISCUSSION

Decisions we have made in designing the approach we introduce are impacted by the evaluations we undertook. We discuss threats to the validity of these evaluations and consider alternatives that could make it easier to apply our approach.

6.1 Threats to Validity

The evaluations of the approach we conducted rely on a data set that focused on six tasks. Although we chose these tasks to be examples of information finding tasks performed by developers, the range of tasks explored is small. By focusing on information finding tasks, we also exclude a significant category of tasks on which developers commonly work, namely coding related tasks. We believe that with minor adaptions, such as tokenizing camel case words or parsing the OCR results to extract in code comments, our approach could be made to work with coding tasks. If we took this approach to coding tasks, the quality of the code base in terms of documentation, naming convention, and so on, could play a large role in the ability of our approach to make accurate predictions. It would be impossible to associate a developer description, or generate a meaningful visual representation, if the code base does not contain descriptive names and lacks documentation. We leave

the investigation of the generalizability of our approach across a wider range of tasks to future study.

Another threat to the findings is the size of the tasks studied and the interleaving of work on different tasks. To fit within a reasonable time frame for a lab setting, the tasks worked on were relatively small in scope. In reality, developers work on complex tasks that can have a huge scope and span multiple topics. In addition, although we caused developers to switch tasks, it is not possible to replicate the many task switches a developer undertakes as he works [17]. It is also unlikely that in reality any single developer would be assigned all six of the tasks we selected at the same time. A field study is likely needed to mitigate these threats.

We also note that the tasks we designed may be more specific in their wording than those that might occur in a developer's normal work pattern. For example, a developer might work on a task in response to some relatively vague verbal request for help from a colleague. In such cases, it is unlikely that the summaries that the developer would write for these tasks are highly descriptive. We mitigate this threat by including a wide variety of low and high quality task summaries written by a group of MT workers with diverse demographic in our evaluation.

While all the developers whose data we collected to create our data set had significant development experience and were actively developing software, all were student developers, and none were employed in a permanent, professional software development position. As such, it is possible that their working habits may differ to some degree from those of professional software developers, effecting the generalizability of our results.

6.2 Limitations of the Approach

A prerequisite for the application of our approach is that the times at which task switches occur must be indicated. To achieve a fully automatic application of our approach, it is necessary that these task switches be detected automatically and with high accuracy. Automatic detection of task segments (i.e., task switches) is a difficult problem (e.g., [18, 21, 32]). While we are optimistic that the techniques to detect task switches will continue to improve, future work should explore the performance of our approach in the absence of knowing task segment boundaries. It may be that missing or erroneously predicting a task switch could lead to degraded performance in our approach in practice.

It is also possible that in practice the vocabulary a developer uses to describe their task does not match exactly with the words commonly found within the content of the task. For example, a developer might use the word "chart" in their task description, yet in the window content of the task the word "graph" might appear prominently instead. Applications of TF-IDF would miss this connection given its focus on exact word matches. Incorporating some notion of semantic similarity into our approach, for example adding word2vec or another model for word embedding, we might be able to enhance task descriptions to also include semantically similar words. More experimentation in a more realistic setting is needed to investigate the impact and need for semantic similarity.

6.3 Artifact Access

Using OCR and capturing a developer's screen content has several benefits. First, it is an application agnostic approach that does not require any instrumentation of applications. As well, a screenshot shows us the *exact* content a developer is looking at in the moment. While OCR performed well for the purposes of our analysis, there are many drawbacks that could limit its usability in practice. For one, OCR is an extremely CPU intensive task. Processing screenshots in real time in the background while a developer works may be impractical for this reason. An obvious alternative might be to send screenshots to the cloud for processing, but privacy concerns, both from the developer's and company's perspective, limit the applicability of this approach. Another issue is the noise generated when using OCR. This may be alleviated to some extent by using a commercial option rather than the open source Tesseract engine. However, we can not guarantee that the product of a screenshot processed with OCR is exactly the same as the content a developer saw on their screen when the screenshot was taken.

An alternative which we will investigate in future work is to track all file accesses and edits made within the scope of a task segment. If we know which files a developer is interacting with, we can extract the contents of the file directly. The benefit of knowing exactly which information in a document is being viewed would be lost in such an approach. However, this loss may be outweighed by the ability to produce cleaner data, and the much lower CPU usage. The contents of web page visits could also be extracted relatively easily with the help of a browser extension. However, it could be difficult to obtain information from applications such as instant messaging and email clients, as there is a much wider range of choices for a developer to use in these cases. For this reason, producing a suite of instrumentations for all the most commonly used applications is impractical. Further investigation is needed to determine how much predictive power is lost by the exclusion of these categories of applications.

6.4 Representations from Window Titles

As mentioned in section 3, in addition to recording screenshots of a developer's active window, our tool also recorded the window title of every window the developer accessed. Unfortunately, due to a recording error window title data was lost for 5 of the 17 developers in our data collection session.

To investigate whether the easier to collect information about application window titles might suffice for supporting our approach, we evaluated RQ1 with the window title data from the 12 developers, in place of the information extracted using OCR. Comparing the results of this evaluation with the results from the same 12 developers using screen content, we found that while the results were lower overall, the difference was modest (64.4% accuracy using window titles vs 70.3% accuracy using screen content). While screen content proved to be a superior choice of data source in almost all cases, window titles seem like a viable alternative especially given the savings in CPU resources. Worth investigating is whether a combination of our approach using window titles and the other data extraction techniques mentioned above can rival the results we achieved using screen content.

7 RELATED WORK

Our approach aids in determining the task driving a segment of work performed by a developer. The association of a task with work provides clues to what and why a software developer is performing the work and is thus related to the *intent* of the developer in undertaking the work. Determining the intent of a developer is a growing area of research. The more we know about a developer's intention, such as the task she is working on, the better we can support the developer, for example by providing better code recommendations (e.g., [5, 13]). Approaches have been developed to determine intent from how a developer interacts with the computer, from the documents produced by a developer and from a mix of both. We describe approaches in each of these categories and also describe related work in finding meaning in artifacts.

Intent from Interactions. For some research systems, intent is specified through specific interactions a developer takes within the environment in which they work. As mentioned, in the Mylyn system, a developer can indicate through an explicit click of the button on which issue they are currently working: the text in an issue provides information about the developer's intent [12]. In the Jasper system, a developer can create special working areas of their development environment into which fragments of work can be placed for later recall [3]. The approach we consider in this paper relieves the developer from a priori indicating work on a specific task.

Other researchers have attempted to determine automatically the higher-level activities developers perform based on their interaction with the computer. For example, Mirza et al. used temporal and semantic features based on window interactions and the window titles over 5 minute time windows to predict one of six work activity categories: writing, reading, communicating, system browsing, web browsing, and miscellaneous [22]. In a controlled lab study and field study with 5 participants, they achieved an accuracy of 81%. Koldijk et al. investigated the predictive power of keyboard and mouse input, as well as application switches and the time of day, for predicting a larger set of 12 high-level task types—such as reading email, programming, creating a visualization—for a given 5 minute period of time [14]. Using classifiers trained on an individual basis, they were able to achieve up to 80% accuracy. However, they found that a classifier trained on one person is highly individual to that person and does not generalize well to other people. In an approach more specific to software developers, Bao et al. explore the use of conditional random fields (CRFs) to predict one of six development activities: coding, debugging, testing, navigation, search, or documentation [2]. Applying their approach to data collected from 10 software developers over a week, the authors found they were able to classify an activity with an accuracy of 73%. The results of Bao et al. point to the difficulty of determining at a fine granularity what a developer is working on at a specific moment. In our work, we aim to determine the content of a developer's task rather than the kind of activity being undertaken, which we see as a complementary goal.

Intent from Documents. Researchers have also looked into the extraction of intent from natural language documents associated with a software development. Early on, researchers have tried to detect the coarse intent of sentences in emails and tried to summarize them, for example to add them to a to do list (e.g., [4]). Di Sorbo et al. introduced the concept of intention mining in the context of emails in software development [33]. They used a Natural Language Processing (NLP) approach to classify the content of development emails according to the purpose of the emails, such as feature request or information seeking. The researchers defined six categories that describe the intent of a developer's sentence and reported a 90% precision and 70% recall for their approach in the context of email intent classification. Huang et al. attempted to generalize the approach of Di Sorbo et al. to developer discussions in other mediums, for example those contained in issue reports [9]. They found that the NLP patterns used did not adapt well to other mediums, achieving an accuracy of only 0.31. By refining the taxonomy of intentions defined by Di Sorbo et al., and applying a convolutional neural network (CNN) based approach, the authors were able to improve on the results of the original paper by 171%. These approaches aim to classify what the content of a document is attempting to state as compared to our approach in this paper which aims to determine what the developer is attempting to do.

Intent from a Combination of Interactions and Documents. Shen et al. [31] use a combination of information about how a user interacts with windows on their screen and email messages the user handles in their TaskPredictor system. Using supervised machine learning, they predict on which task a user is working. However, this techniques requires the user to pre-define the tasks on which they work so that they can be predicted and the classifier needs to be trained on some of the user's data beforehand. Our approach differs in assessing methods for representing the work being performed based on information that a developer works on through screen scraping; these representations can be used for predicting which of a known set of tasks the work represents and for generating word cloud representations of the work that a developer can recognize irrespective of having a set of known tasks.

Finding Meaning in Artifacts. The content of artifacts created as part of, or about, software development contain significant meaning. Software engineering researchers have developed techniques to find particular meaning in artifacts that have similar characteristics to the approach we develop in this paper. For example, Ponzanelli et al. present CodeTube, an approach that mines video tutorials from the web to enable developers to query the contents of the tutorial to retrieve relevant fragments [28]. The authors used OCR and speech recognition in order to extract text from the videos and evaluate the relevancy of fragments to the user's query. The determination of what a segment of video is about is similar to the problem we tackle of what a segment of a developer's work is about.

8 SUMMARY

Have you ever wondered what you worked on throughout a day, possibly to record time spent on different projects? Have you ever wanted to look back and find where you worked on a particular task to find what resources you consulted as part of the task?

This paper introduces and evaluates an approach to help support these goals. Given knowledge of task boundaries, which is possible from using automated task switch detection techniques, our

approach extracts the contents of the active window being worked with on a regular basis, uses optical character recognition (OCR) and word tokenization to transform the contents into tokens and words, and applies TF-IDF to form a vector representation of the task segment. On information seeking tasks, we showed that this vector representation can help identify which task a segment represents for a known set of tasks with an averaged accuracy of 70.6%.

We also investigated the production of word cloud representations of a task segment using TF-IDF scores for each word in a bag of words formed from the screen content of the task segment as a developer worked. Through a preliminary evaluation, we found that participants could determine which task a word cloud for a segment of work represented with reasonable accuracy (67.9% on average) for several information-seeking tasks. Interestingly, the accuracy rose only modestly when considering identifying the task based on a word cloud formed from all segments comprising work on a task.

This approach shows promise for helping to determine automatically the task on which a developer is working during different time periods. When the task can be identified, various tools can be improved that a developer relies upon and new tools can be introduced to help support such activities as time tracking. Future work can investigate how the approach we introduce applies to a broader set of kinds of tasks performed by a developer.

ACKNOWLEDGEMENTS

We thank all our study participants and the reviewers for their helpful feedback. This work was funded, in part, through a collaborative grant with ABB Inc. (Industrial Research Grant F17-05273-22R77358), supported by NSERC (CRDPJ 530226-18).

REFERENCES

[1] Anonymous. 2020. Supplemental Material for the paper "Identifying and Describing a Software Developer's Tasks". https://doi.org/10.5281/zenodo.3764485

[2] Lingfeng Bao, Zhenchang Xing, Xin Xia, David Lo, and Ahmed E. Hassan. 2018. Inference of development activities from interaction with uninstrumented applications. *Empirical Software Engineering* 23, 3 (June 2018), 1313–1351. https://doi.org/10.1007/s10664-017-9547-8

[3] Michael J. Coblenz, Andrew J. Ko, and Brad A. Myers. 2006. JASPER: An Eclipse Plug-in to Facilitate Software Maintenance Tasks. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology Exchange*. Association for Computing Machinery, 65–69. https://doi.org/10.1145/1188835.1188849

[4] Simon Corston-Oliver, Eric Ringger, Michael Gamon, and Richard Campbell. 2004. Task-focused summarization of email. In *Text Summarization Branches Out*. Association of Computational Linguistics, 43–50.

[5] K. Damevski, H. Chen, D. C. Shepherd, N. A. Kraft, and L. Pollock. 2018. Predicting Future Developer Behavior in the IDE Using Topic Models. *IEEE Transactions on Software Engineering* 44, 11 (2018), 1100–1111. https://doi.org/10.1109/TSE.2017.2748134

[6] Márcio Kuroki Gonçalves, Cleidson RB de Souza, and Victor M Gonzalez. 2011. Collaboration, Information Seeking and Communication: An Observational Study of Software Developers' Work Practices. *J. UCS* 17, 14 (2011), 1913–1930.

[7] Victor M González and Gloria Mark. 2004. "Constant, Constant, Multi-tasking Craziness": Managing Multiple Working Spheres. In *Proceedings of the 2004 Conference on Human Factors in Computing Systems, CHI 2004*. 113–120.

[8] Thomas Gottron. 2009. Document Word Clouds: Visualising Web Documents as Tag Clouds to Aid Users in Relevance Decisions. In *Research and Advanced Technology for Digital Libraries*, Maristella Agosti, José Borbinha, Sarantos Kapidakis, Christos Papatheodorou, and Giannis Tsakonas (Eds.). Springer Berlin Heidelberg, 94–105.

[9] Qiao Huang, Xin Xia, David Lo, and Gail C. Murphy. 2018. Automating Intention Mining. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2876340 Early access.

[10] ImageMagick. 2020. ImageMagick. https://imagemagick.org/index.php. [Accessed March 5, 2020].

[11] Tom Kenter, Alexey Borisov, and Maarten De Rijke. 2016. Siamese cbow: Optimizing word embeddings for sentence representations. *arXiv preprint arXiv:1606.04640* (2016).

[12] Mik Kersten and Gail C. Murphy. 2006. Using Task Context to Improve Programmer Productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*. ACM, 1–11. https://doi.org/10.1145/1181775.1181777

[13] Katja Kevic and Thomas Fritz. 2017. Towards Activity-Aware Tool Support for Change Tasks. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 171–182.

[14] Saskia Koldijk, Mark van Staalduinen, Mark Neerincx, and Wessel Kraaij. 2012. Real-time task recognition based on knowledge workers' computer activities. In *Proceedings of the 30th European Conference on Cognitive Ergonomics (ECCE '12)*. Association for Computing Machinery, 152–159. https://doi.org/10.1145/2448136.2448170

[15] Mary L. McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia Medica* 22, 3 (Oct. 2012), 276–282.

[16] André Meyer, Gail C Murphy, Thomas Zimmermann, and Thomas Fritz. 2017. Design Recommendations for Self-Monitoring in the Workplace: Studies in Software Development. *PACM on Human-Computer Interaction* 1, CSCW (2017), 1–24. https://doi.org/10.1145/3134714

[17] A. N. Meyer, L. E. Barton, G. C. Murphy, T. Zimmermann, and T. Fritz. 2017. The Work Life of Developers: Activities, Switches and Perceived Productivity. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1178–1193. https://doi.org/10.1109/TSE.2017.2656886

[18] A. N. Meyer, C. Satterfield, M. Züger, K. Kevic, G. C. Murphy, T. Zimmermann, and T. Fritz. 2020. Detecting Developers' Task Switches and Types. *IEEE Transactions on Software Engineering* (2020). Early access.

[19] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[20] Allen E. Milewski. 2007. Global and task effects in information-seeking among software engineers. *Empir. Softw. Eng.* 12, 3 (2007), 311–326. https://doi.org/10.1007/s10664-007-9036-6

[21] Hamid Turab Mirza, Ling Chen, Gencai Chen, Ibrar Hussain, and Xufeng He. 2011. Switch detector: an activity spotting system for desktop. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM '11)*. Association for Computing Machinery, 2285–2288. https://doi.org/10.1145/2063576.2063947

[22] Hamid Turab Mirza, Ling Chen, Ibrar Hussain, Abdul Majid, and Gencai Chen. 2015. A Study on Automatic Classification of Users' Desktop Interactions. *Cybernetics and Systems* 46, 5 (2015), 320–341. https://doi.org/10.1080/01969722.2015.1012372

[23] Mozilla. 2020. Bugzilla. https://bugzilla.mozilla.org/home. [Accessed August 31, 2020].

[24] NLTK. 2020. Natural Language Toolkit (NLTK). https://www.nltk.org/. [Accessed March 5, 2020].

[25] Nuria Oliver, Greg Smith, Chintan Thakkar, and Arun C Surendran. 2006. SWISH: semantic analysis of window titles and switching history. In *Proceedings of the 11th International Conference on Intelligent User Interfaces*. 194–201.

[26] Chris Parnin and Robert DeLine. 2010. Evaluating cues for resuming interrupted programming tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. Association for Computing Machinery, 93–102. https://doi.org/10.1145/1753326.1753342

[27] Chris Parnin and Spencer Rugaber. 2011. Resumption strategies for interrupted programming tasks. *Software Quality Journal* 19, 1 (March 2011), 5–34. https://doi.org/10.1007/s11219-010-9104-9

[28] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Barbara Russo, Sonia Haiduc, and Michele Lanza. 2016. CodeTube: Extracting Relevant Fragments from Software Development Video Tutorials. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. 645–648.

[29] RAKE. 2020. Python implementation of the Rapid Automatic Keyword Extraction algorithm using NLTK. https://pypi.org/project/rake-nltk/. [Accessed March 5, 2020].

[30] Stuart Rose, Dave Engel, Nick Cramer, and Wendy Cowley. 2010. Automatic Keyword Extraction from Individual Documents. In *Text Mining: Applications and Theory*. 1–20. https://doi.org/10.1002/9780470689646.ch1 Journal Abbreviation: Text Mining: Applications and Theory.

[31] Jianqiang Shen, Werner Geyer, Michael Muller, Casey Dugan, Beth Brownholtz, and David R Millen. 2008. Automatically finding and recommending resources to support knowledge workers' activities. In *Proceedings of the 13th International Conference on Intelligent User Interfaces (IUI '08)*. Association for Computing Machinery, 207–216. https://doi.org/10.1145/1378773.1378801

[32] Jianqiang Shen, Lida Li, and Thomas G. Dietterich. 2007. Real-time detection of task switches of desktop users. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence (IJCAI'07)*. Morgan Kaufmann Publishers Inc., 2868–2873.

[33] Andrea Di Sorbo, Sebastiano Panichella, Corrado A. Visaggio, Massimiliano Di Penta, Gerardo Canfora, and Harald C. Gall. 2015. Development Emails Content Analyzer: Intention Mining in Developer Discussions (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 12–23. https://doi.org/10.1109/ASE.2015.12

[34] Tesseract. 2020. Tesseract Open Source OCR Engine. https://github.com/tesseract-ocr. [Accessed March 5, 2020].